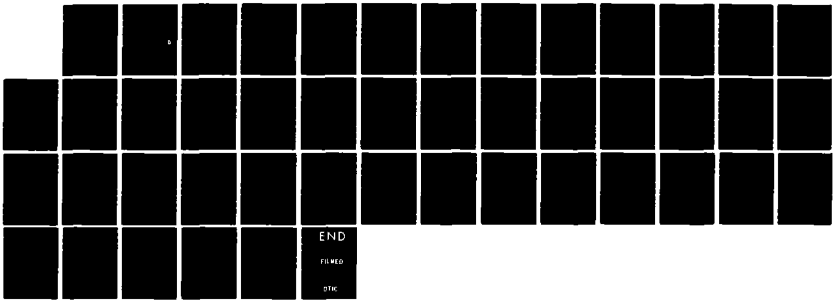
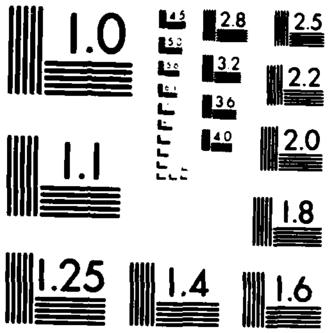

END
FILMED
DTIC


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

(12)

KES.U.84.6

Synthesis of Tree-Structured Computing Systems Through Use of Closures

by

Richard M. King

Cordell Green
Principal Investigator

Kestrel Institute
1801 Page Mill Road
Palo Alto, CA 94304

FINAL TECHNICAL REPORT

November 1984

Prepared for:

Air Force Office of Scientific Research
Building 410
Bolling AFB, DC 20332

Research sponsored by the Air Force Office of Scientific Research (AFSC), United States Air Force, under contract F49620-82-C-0007. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): KES.U.84.6	5. MONITORING ORGANIZATION REPORT NUMBER(S): AFOSR-TR- 85-0065
6a. NAME OF PERFORMING ORGANIZATION: Kestrel Institute; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION: Air Force Office of Scientific Research
6c. ADDRESS (City, State and ZIP Code): 1801 Page Mill Road, Palo Alto CA 94304	7b. ADDRESS (City, State and ZIP Code): Directorate of Mathematical & Information Sciences, Bolling AFB DC 20332-6448
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: AFOSR; 8b. OFFICE SYMBOL (If applicable): NM	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: F49620-82-C-0007
8c. ADDRESS (City, State and ZIP Code): Bolling AFB DC 20332-6448	10. SOURCE OF FUNDING NOS.: PROGRAM ELEMENT NO. 61102F; PROJECT NO. 2304; TASK NO. A2; WORK UNIT NO.

11. TITLE (Include Security Classification): SYNTHESIS OF TREE-STRUCTURED COMPUTING SYSTEMS THROUGH USE OF CLOSURES,

12. PERSONAL AUTHOR(S): Richard King

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Final	FROM 1/1?/83 TO 30/8/84	29 NOV 84	40

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	Multi-processor synthesis; tree-structured multiprocessors; concurrency; closures; divide and conquer; trees; actors.

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

During this past year the investigators have concerned themselves with the synthesis of tree structures. These structures offer, in the opinion of the investigators, the best hope of achieving subpolynomial running times for typical problems without a degree of interconnection that makes physical implementation difficult.

One would like to be able to synthesize trees using divide and conquer. Divide and conquer is an appealing technique for tree synthesis because of the isomorphism between the shape of the desired synthesized system and the recursive descent implicit in divide and conquer. Additionally, the technique makes good use of theorem proving techniques which are rapidly being developed for other purposes. Certain problems arise, however, when one tries to use divide and conquer to synthesize a tree-structured computing system. The exact characteristics of the problems that can arise fall into three categories, to be described below, but the basic difficulty is that nodes that are high in the tree are required to either (CONT.)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Dr. Robert N. Buchal	22b. TELEPHONE NUMBER (Include Area Code): (202) 767- 4939; 22c. OFFICE SYMBOL: NM

DD FORM 1473, 83 APR — EDITION OF 1 JAN 73 IS OBSOLETE. — UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

ITEM #19, ABSTRACT, CONTINUED: compute or communicate large amounts of data.

The investigators' primary solution to this problem is to replace the original specification which in general declares the existence of an output array that depends on various elements of the input array, into an equivalent specification which declares the existence of a certain closure, or specialized functional object, together with a declaration that it be applied. Constraints are imposed on the closure so that application of this closure will have the desired effect. The investigators show that closures can be computed and applied rapidly, in time $O(\log^i n)$ for small, constant i on problems of size n, even in many cases where the normal results of divide and conquer would be a computation that could only be performed in time $O(n^j)$ for strictly positive constant j.

The investigators have also found an interesting synthesis path for several binary addition circuits that uses this technique and another technique called quantifier levelling.

Contents

Plates

Abstract

During this past year we have concerned ourselves with the synthesis of tree structures. These structures offer, in our opinion, the best hope of achieving subpolynomial running times for typical problems without a degree of interconnection that makes physical implementation difficult.

One would like to be able to synthesize trees using divide & conquer. Divide & conquer is an appealing technique for tree synthesis because of the isomorphism between the shape of the desired synthesized system and the recursive descent implicit in divide & conquer. Additionally, the technique makes good use of theorem proving techniques which are rapidly being developed for other purposes (see [Smith-83]). Certain problems arise, however, when one tries to use divide & conquer to synthesize a tree-strucured computing system. The exact characteristics of the problems that can arise fall into three categories, to be described below, but the basic difficulty is that nodes that are high in the tree are required to either compute or communicate large amounts of data.

Our primary solution to this problem is to replace the original specification, which in general declares the existence of an output array that depends on various elements of the input array, into an equivalent specification which declares the existence of a certain *closure*, or specialized functional object, together with a declaration that it be applied. Constraints are imposed on the closure so that application of this closure will have the desired effect. We show that closures can be computed *and applied* rapidly, in time $O(\log^i n)$ for small, constant i on problems of size n, even in many cases where the normal results of divide & conquer would be a computation that could only be performed in time $O(n^j)$ for strictly positive constant j.

We have also found an interesting synthesis path for several binary addition circuits that uses this technique and another technique called *quantifier levelling*.

[illegible]

Chapter 0

Trees of Processors

In this report we examine one class of methods for producing highly concurrent architectures. These architectures are vital to meet the needs for sufficiently fast computation to make certain problems practical. Automatic systems for the synthesis of these architectures are therefore important because hand crafting is a difficult, expensive and error-prone process. In this report we explore the synthesis of tree-structured architectures. Other architectures have been explored in prior reports ([King-83], [KingBrown-82]).

Trees of processors can be used to efficiently implement many specifications because the tree is that topology with fixed arity and lowest connectivity that allows a distinguished node to have contact with all other nodes in $O(\log n)$ steps, which is clearly the best possible. TRANSCONS ([King-83]) therefore has facilities for specifying, synthesizing and manipulating trees.

The description of a tree is specified in TREE declarations, described below. Before describing the syntax of a TREE declaration, we will describe some of the semantics we intend for it.

The trees we intend to address are used to shorten the longest path lengths within the collection of processors, and to balance the workload of a computation. There are problems amenable to a tree solution, portions of which are in some sense more important than others (for example Optimal Binary Search Trees), but in these problems there must be a specification of relative importance that has a size comparable to the size of a good specification of the solution. We will therefore model solutions to problems of this sort by building separate trees and AGGREGATEing them. Each tree described in a single locution will be balanced.

Several principles govern the design of the tree system of TRANSCONS.

- All trees are as balanced as possible. (We use binary trees; extensions to trees of higher arity introduce no new principles.) *No flexibility in terms of shape is assumed, nor is any way provided for expressing shapes.*
- A tree specification must include a size, which can be any integer greater than one.
- The shapes of two trees of the same size are identical. That is, there is an isomorphism $\simeq$ between two trees of the same size that maps parents, left children and right children respectively into parents, left children and right children. There are "compile-time" constructs in the TRANSCONS language that allow for the specification of connections to the node that is $\simeq$ to a given node, or AGGREGATION between corresponding nodes of different trees. One way to achieve this identity of shape is to have a left-biased tree that is as balanced as

possible. In other words, path lengths from root to leaves differ by at most one and if one such path is longer than a second the first path must be to the left of the second.

- The nodes of a tree are divided into three groups. They are the root, the internal nodes, and the leaves. The leaves are further distinguished by indices. References to any of these classes of tree nodes, either to attach procedure, to specify communication such as **HEARS**, or to **AGGREGATE** can be made. Tags are provided for a node to refer to a node of another tree that is $\simeq$ to it if the two trees are the same size. This allows nodes in $\simeq$-equivalence classes to be **AGGREGATED** or to **HEAR** each other. For this to work values have to be declared properly. Note that a leaf has to offer instances of values that are **HEARd** upward, and the root has to offer values that are **HEARd** downward.

To support these stipulations we have the **TREE** data type. A tree is declared and its components laid out using the type facility of CHI. As an example, we will describe below a situation where there are two trees, T and U. Each is of size n. Each internal node of T passes a value to its children after having multiplied it by a value from the corresponding internal node of U. Each internal node of U adds values from its two children. The procedures at the leaves of T and U, respectively, are described by functions H and G, not interpreted here.

```
T istype TREE (i), i∈[1 ..n−1] size n
     root HAS  v TALKS leftson  (SENDS v)
                 TALKS rightson (SENDS v)
                 HEARS source (USES outside-value)
                 HEARS U.root (USES u-value)
     inter HAS v TALKS leftson  (SENDS v)
                 TALKS rightson (SENDS v)
                 HEARS parent (USES v.parent)
                 HEARS U.inter (USES u-value)
     leaf HAS  l_i HEARS parent (USES v.parent)
U istype TREE (i), i∈[1 ...n−1] SIZE n
     root HAS  u TALKS T.root (SENDS u)
                 HEARS  leftson(USES v.left)
                 HEARS rightson(USES v.right)
     inter HAS u TALKS T.inter (SENDS u as u-value)
           HAS v TALKS parent  (SENDS v)
                 HEARS  leftson(USES v.left)
                 HEARS rightson(USES v.right)
     leaf HAS  v TALKS parent  (SENDS v)
                 HEARS some_i  (USES A_i)
(in T.root)
     v ← outside-value × u-value
(in T.inter)
     v ← v × u-value
(in T.leaf_i)
     l_i ← H(v)
(in U.root)
     v ← v.left + v.right
(in U.inter)
     v ← v.left + v.right
     u ← v
(in U.leaf_i)
```

$v \leftarrow G(A_i)$

Note the SENDS *u* as *u-value* locution. This causes a value to be known as *u* in the the intermediate node but to be known as *u-value* in the recipient.

In the next Chapter we show the power and limitations of divide & conquer and describe a technique for mitigating the limitations. In the Chapter following that we give examples of the use of this technique.

Chapter 1

Closure-Assisted Divide & Conquer or, LAMBDA: The Ultimate Transceiver[*]

§1.1 Motivation

Suppose information must flow from processor B to processor A, but there is a conceptual advantage to viewing the problem as if information were flowing the other way. We have two motivating situations where this is the case. One is the handshake problem, where an intermediate processor in a chain of pipelining processors must be able to declare its readiness to handle another datum after it has proccessed a first. The second is problems requiring tree-structured collections of interconnected processors. We would like to use divide & conquer to synthesize these trees, but that technique is difficult to apply if data conceptually flow both up and down the tree. It becomes easier if the flow is conceptually one way. We claim that divide & conquer is a powerful synthesis technique that can produce a large class of tree structured architectures if problems can be rephrased in terms of one-way data flow.

We want to bring about a structure in which information flowing in one direction tells the receiving processor what to do with other information computed in the receiving processor. We want a new type of datum, the "self-addressed stamped envelope". Processor A sends processor B an instance of this type of datum, and B can later use it to cause the data to be sent back to A and to be used properly.

We use *closures* to do this. We explore the weaknesses of divide & conquer without closures below, and then we explore some of the implications of closures.

§1.2 Divide & Conquer Paradigm and Tree Synthesis

Divide-and-conquer (D&C) is a widely used technique for the synthesis of single-processor programs, and one feels that it should be a good technique for the synthesis of tree-shaped parallel structures. Trouble often arises, however, when we try to use D&C for this purpose.

[*] With apologies to Guy Steele [Steele-77]

Consider what the D&C technique actually is. "To solve a 'large' problem instance, break it into pieces, solve the problem for each of the pieces, and combine the solutions". This is a technique for generating $O(n)$ and $O(n \log n)$ time, single processor solutions to a wide variety of problems. See, for example, [**Smith-83**] and [**Knuth-volI**].

Intuition would lead us to believe that D&C is useful for synthesizing tree-structured parallel structures, because the structure of a solution closely matches the structure of the set of processors. Three sorts of problems arise, however:

- **rootlock**: When we try to combine two subproblems' solutions, the amount of information traveling either from one subproblem to the other or from the subproblems to the combination operator, or the amount of work necessary to combine, may be asymptotically large in the problem size. A naïvely synthesized parallel structure would have to perform all of this work in one processor, namely a "root" processor that has responsibility for combining two half-solutions into a solution to the whole problem.
- **sequentiality**: In a variant of D&C, one solves one of the subproblems *first*, and uses some function of the solution as a parameter to the process that takes place on the second side. It is clear that in this case no problem element can enter the computation until all previous elements have been used. There is no concurrency.
- **bidirectionality**: Information might have to flow both up and down the tree to make a solution. This situation can make formal description of a combination operator for D&C hard. It might appear that this condition is intrinsic to divide & conquer, but that is not the case. The data could already be distributed among an array of processors (or available to be so distributed) and the division step can manipulate indices only.

It is possible to have bidirectionality without sequentiality, but not *vice versa*. Rootlock is independent of the other two situations.

These three properties of D&C solutions to specifications are impediments to easy synthesis of tree-structured parallel structures for these specifications.

A specification, three of whose natural D&C solutions have one of these features each, is Prefix Summation. In this specification we have a vector A of dimension n, and we want to create a vector A' such that $\forall 1 \leq i \leq n[a'_i = \sum_{1 \leq j \leq i} a_j]$. In what follows I will use the words "left" and "right" as if the array were arranged in a row with a_1 leftmost and a_n rightmost.

One solution is "to perform prefix summation on a non-trivial vector, divide it into two halves, perform prefix summation on each half, and add the rightmost element of the left result to each element of the right result". This solution has two-way data flow.

A second solution is to first define "augmented prefix summation with augend z" as $\forall 1 \leq i \leq n[a'_i = z + \sum_{1 \leq j \leq i} a_j]$. We then say that to perform augmented prefix summation with augend z on a non-trivial vector $a_{l:u}$, divide it into two halves $a_{l:u'}$ and $a_{u'+1:u}$, perform augmented prefix summation with z on the left half, and perform augmented prefix summation with $z + a'_{u'}$ on the right half. This is intrinsically sequential.

A third solution is similar to the first, except that the result vector is carried up the tree as the value of the D&C step rather than having as the goal to develop the new values at the leaves. This has rootlock, i.e., it is intrinsically an $O(n)$ solution, as it requires funnelling the entire result vector through the root.

Our solution to this problem is to use an upward (toward the root) flow of *closures* to represent the downward flow of data.

The solution is based on the idea of passing a form of data called a *closures* up the tree. A closure is a procedure or function definition together with an environment, i.e., a set of name/value pairs. When a closure is invoked, the procedure or function is invoked in the included environment as augmented by parameter binding. When processor A passes processor B a closure, A is said to be the closure's *host* and B the *recipient*.

The actual closure is not sent. Instead, a token is sent that the recipient can use to invoke the closure's program by sending back (to the host) the token together with values for the arguments. By convention this causes the host to invoke the procedure, using stored bindings and possibly some new ones from the sent arguments as an environment. The motivation for this is that *while conceptually data (i.e., the closures) are flowing in only one direction, in fact data are flowing in the other direction as well (in the form of arguments and invocation requests).*

In this manner we can reformulate the problem from one of creating some new array that is a function of an existing array to that of creating a closure that, when invoked, will perform a given action on the leaves of a tree. This action is the creation of an *element* of the new array in each leaf. The original specification is transformed into a specification that declares the existence of a closure that, when invoked, will satisfy the original specification, followed by a specification that the new closure be invoked. The three barriers to simple tree solutions described above do not arise. We consider a synthesis of parallel prefix summation in the next Chapter.

We have exchanged the difficulty of reasoning about two-way data flow with the need to reason about closures. We feel that this is a good bargain because reasoning about closures only requires the addition of new axioms to a theorem prover's data base, while two-way data flow requires changes in the way we look at D&C. Below we show that this change of view costs little speed, and in the next Chapter we show that no expressive power is lost.

We conjecture that this technique can bring most $O(\log n)$ and $O(\log^2 n)$ tree parallel structures within the reach of a D&C-based synthesis method. We support this conjecture by several synthesees in the next Chapter. Since a tree-structured processor is inexpensive to manufacture compared to more highly interconnected machines and seems to be reasonably powerful, we feel that automatic tools that make use of this power easier would be an important contribution to the technology of synthesis of parallel structures.

We first prove that the computation of the closure in the root node is fast:

Theorem 1.1. *Suppose a problem fits a divide and conquer scheme without sequentiality or bidirectionality. That is, that the computation of the result in question for the substring of the problem ranging from l to u is*

$$V_l^u = \begin{cases} \textbf{if } l = u \textbf{ then } V_l' \\ \textbf{otherwise } G(V_l^{u'}, V_{u'+1}^u) \end{cases}$$

and $T(G)$ (the time to compute G) is $\leq O(F(u-l+1))$, where F is a nondecreasing function. Then $T(V_1^n) = O(F(n)\log n)$.

Proof: Note that the form of the definition of V_l^u precludes sequentiality and bidirectionality. We are using value semantics for the call to G.

$T(V_i^i) = T(V')$, so $T(V')$ is bounded. Say $T(G) \leq c_0 F(u-l+1)$. We offer an inductive proof that $T(V_l^u) \leq c_0 F(u-l+1)\lg(u-l+1) + T(V')$, where c_0 is the constant of $T(V_1^n) = O(F(n)\log n)$.

The base case is immediate.

If $l \neq u$ then

$$\begin{aligned} T(V_l^u) &= \max(T(V_l^{\lfloor (l+u)/2 \rfloor}), T(V^u_{\lceil (l+u+1)/2 \rceil})) + T(G) && \text{(definition, nonsequentiality)} \\ &\leq c_0 F((u-l+1)/2) \lg((u-l+1)/2) + T(V') + T(G) && \text{(by induction)} \\ &\leq c_0 F(u-l+1) \lg(u-l+1) + T(V') && \text{(monotonic } F\text{)} \end{aligned}$$

This is $O(F(u-l+1)\log(u-l+1))$, which is $O(F(n)\log n)$ at $T(V_1^n)$. ∎

This theorem only holds if sequentiality and bidirectionality are not present. Sequentiality can not be present because $T(V_l^u) = \max(T(V_l^{\lfloor (l+u)/2 \rfloor}), T(V^u_{\lceil (l+u+1)/2 \rceil})) + T(G)$ only holds if the computation of the V's can proceed in parallel, and bidirectionality must not be present as there is nothing in the statement of the theorem to allow for this. It holds even if rootlock is present, but in such a case the theorem produces a weak result, since $F(n)$ would be large.

We then prove that the application of the closure that is computed in the root is also fast:

Theorem 1.2. *Suppose a closure is computed in the root of a balanced binary tree. That closure can contain closures whose hosts are its children. Those closures, in turn, can contain closures whose hosts are their children, etc. Suppose all closures computed within the tree are of the form* $C_l^u = \lambda_z^{\vec{V}_l^{u'}, \vec{V}^u_{u'+1}, \vec{W}_l^u}[G(\vec{V}_l^{u'}, \vec{V}^u_{u'+1}, W_l^u)]$ *where* $\vec{V}_l^u$ *includes* C_l^u *and values that are available in constant time,* W_l^u *includes locally available values, and* G *is of the form* $G(\vec{V}_l, \vec{V}_r) = (C_l(G_l(\vec{V}_l, \vec{V}_r)) \parallel C_r(G_l(\vec{V}_l, \vec{V}_r \parallel G_0(\vec{V}_l, \vec{V}_r, W_l^u))))$ *(here* C_l *is the closure contained in* $\vec{V}_l$ *and* G_0 *can affect* W_l^u*.) If* $\max(T(G_\bullet), T(G_l), T(G_r)) = O(F(l-u+1))$*, then* $T(C_1^n) = O(F(n)\log n)$

Proof: virtually identical to previous proof. ∎

In summary, the technique of computing closures from component closures is a technique which, together with divide-and-conquer, provides the ability to synthesise a wide variety of tree structures with few of the technical problems that other synthesis methods might encounter concerning reasoning about path lengths or the cardinality of sets of nodes. It allows us to do this and to still produce the $O(\log n)$ (or $O(\log^i n)$ for small i) parallel structures we expect from trees.

§1.3 Description of Closures

A closure consists of a procedure, and bindings for some of the procedure's free variables. The procedure, in turn, consists of a piece of program and a binding list. The concept was first described in Church's λ-calculus [Church-51]. Closures are valued for their expressive power even on single-processor algorithms. They are elements primarily of dialects of LISP. See, for example, [Steele-77], [Moon-82], [Interlisp-83]. A similar concept, *actors*, is also found in other languages (See, for example, PLASMA in [SmiHew-75].) Actors are also described, as here, as a method of expressing interprocessor communications concepts. I here explore a case in which it makes the task of writing programs an easier one for computers.

It is common to use the notation $\lambda x_1, x_2, \ldots, x_n[F(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m)]$ to denote abstraction of a function of n parameters from a function of $n+m$ parameters. The y_i's are free variables, meaning that their values are determined by some of the context in which the function is evaluated.

We will use "$\lambda^{y_1,\ldots}_{x_1,\ldots}[F(x_1, \ldots, y_1, \ldots, z_1, \ldots)]$" to denote a piece of program text that makes a closure that can be applied to as many parameters as there are x's. In other contexts we will use that to name the closure itself. When it is applied the x-values from the application, the y-values available at closure creation time and the z-values at application time will be used. The y's are called the *closed variables*. We will use $\lambda^{y_1=v_1,\ldots}_{x_1,\ldots}[F(x_1, \ldots, y_1, \ldots, z_1, \ldots)]$ created by the above fragment to denote the closure in which $y_1{=}v_1, \ldots$.

§1.4 Transmitting a Closure

To transmit a closure from one processor to another, it is *not* necessary to transmit the entire program and all of the environment values, provided that the processor sending the closure stands willing and able to perform the work, and that the side effects are within reach of the sending processor. In the cases we explore there are side effects that are *only* within reach of the sending processor. Since the motivation for closures in the first place was the desire for a datum that, when exercised, would cause certain desirable behavior in the host, this will normally be the case.

All that is necessary is that the transmitting processor send a token of some sort. The receiving processor can save the token and later use the closure by sending back the arguments, the token, and control information.

When this is done, the processor sending the closure (and willing to do the work) is called the closure's *host*, and the receiving processor (which has a license to use the closure) is called the *recipient*.

We say that a closure is *live* if there is a possibility that it will be invoked at a given time. A closure becomes live when it is sent and remains live until the recipient reaches a point in its procedure past which it can not invoke the closure. We will have more to say about issues concerning the liveness of closures during the remainder of this Section.

Closures can be efficiently implemented in a reasonable machine model. Internally, a closure can be implemented as a block of memory locations containing a "pointer" to the program fragment and a list of all closed variables and the corresponding values. A pointer to the block could be used as the token. When a closure is applied the recipient can send the host a copy of the token, together with whatever other information is needed (primarily the argument(s)). The host can use the received token and can invoke the proper code with the proper environment and with the arguments bound to the parameters by using the information contained in the closure and message. A piece of program text (in the host processor) that creates a closure will be called a *closure generating form* or CGF, and a piece of text (in the receiving processor) that invokes one will be called a *closure invoking form* or CIF. The class of closures generated by one CGF is a *family*. An instance of the family of closures generated by a specific CGF named C will be called a C *instance* or an *instance from* C. Members of a family differ only in the environments, since the code will be the same.

The required data transmission can be reduced in cases where it is possible to infer various things about the use of a closure. For example, if it is known that only one instance from a given CGF is live at a time, the host needs not send the token, but only the name of the CGF. That name would not vary and can be "assembled into" the CIF. This can be true even if there can be several CGF instances for a given CGF, provided that the host knows what order the recipient will use the closures it receives. If there is only one CGF in a processor, and only one instance of the closures that it generates can be live at one time, the token can vanish; the fact that the "receiving" processor wants to apply any closure is information enough! The closure has been completely swallowed up; information *only* travels from the recipient to the host, even though the synthesis was performed as if data flowed only in the other direction.

A further simplification, of interest for the problem of synthesizing parallel structures that will later be reduced to VLSI, is available. Suppose the following conditions are met: Applying a closure does not include changing state in the host processor. (In this case, for the application to be useful it must cause other applications in the host.) Assume also that there is only one live closure in a given family at any time. Assume further that the values used in that closure to call other closures hosted elsewhere can be computed, using only values available to the host, by means of combinatorial logic (the code fragment is loop-free and consists only of operators chosen from a library of integrateable operators).

In this case it is possible to perform the closure using only "combinatorial logic" in the host processor. Specifically, no register need be provided to hold the closure's parameter in the host processor. Instead, logic must be provided to map a signal representing an application of the closure to signal(s) representing application(s) of the subsequently called closure(s). Registers are provided to hold all of the values of the closure. An example takes from the Parallel Prefix structure (whose derivation sketch is in the next Chapter) will make this clear.

We have the code fragment to synthesize a closure, namely $\lambda_z^{C_l, C_r, v_l}[C_l(z) \parallel C_r(v_l + z)]$[1]. Here it can be established that there is only one outstanding instance of the closure at any time, that the closure does nothing more than apply other closures to a function of its argument, and that the computation performed on the argument is "easy". We can therefore use the circuit of Figure 1:

[1]For clarity, the exposition assumes that the prefix operation is addition, and that we consider addition to be integrateable.

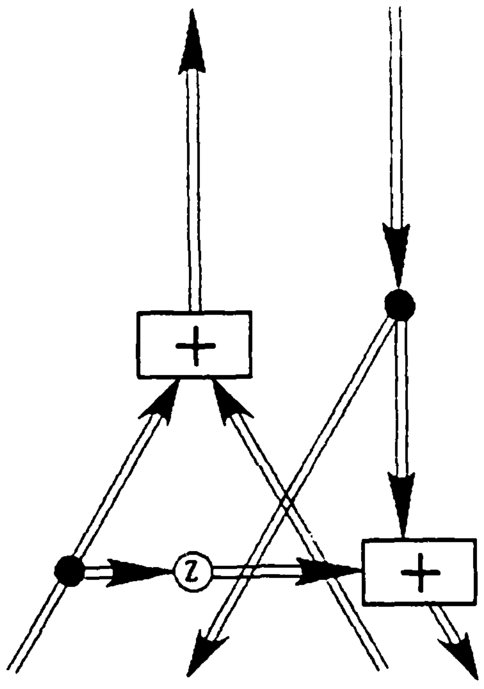


Figure 1. Simplified Parallel Prefix Internal Node

§1.5 Formal Arguments for the Admissibility of Closures

In this Section we formalize the notions we use to argue that restricting communication to the upward direction in trees is a harmless restriction, not preventing the synthesis of tree parallel structures to meet any specification that could have been met absent this restriction, provided only that we also allow upward communication closures and that we not consider the application of a closure that was communicated upward to be a downward communication.

First we need a formal definition of a tree parallel structure:

Definition 1.3. *A* tree parallel structure *(tree structure) is a collection of processors together with programs that meet all of the following conditions:*

- *There are three types of node: leaves, interior nodes (which need not be present), and the root.*
- *(tree) There are various two-way connections ("wires") between nodes as follows: roots have a* left *and* right *wire; interior nodes have a* left, right, *and* parent *wire, and leaf nodes have a* parent *wire. A parent wire must be connected to either a left or a right wire, and vice versa. Node* A *(resp.* B*) is an* ancestor *(resp.* descendant*) of the other if there is a path from it to the other using only left or right → parent (resp. parent → left or right) wires. If the first wire on the path to a descendant is a left (resp. right) wire the descendant is a* left *(resp.* right*) descendant.*

- *(numbered leaves) The leaves are indexed by a totally ordered index set ("numbered") so that the index of one leaf must be less than the index of a second leaf if there is a common ancestor for which the first leaf is a left descendant and the second a right descendant.*
- *(homogeneous) All nodes of one type run the same program. Programs are allowed to do reasonable forms of computation and to try to send and receive information on the wires.*
- *(singly buffered) If a program tries to receive information over a given wire it will do nothing else until the program of the node at the other end of the wire tries to send. If a program tries to send on a wire twice without the other program having tried to receive, the sending program will do nothing else until the other program tries to receive. Programs may perform closure application with no regard to these restrictions, but the transmission of the closures must have obeyed these conditions. Programs may test whether a line has or can accept data and therefore avoid waiting if it can't. The situation where neither program at either end of the wire can send or receive is possible, but only for a bounded amount of time.*

We need a definition of a tree parallel structure with upward communication only:

Definition 1.4. An upward tree parallel structure *is a tree parallel structure in which no communication is specified from any left or right end of a wire to the corresponding parent end. Closure application does not count as a communication.*

This is a formal definition of the objects described by **TREES** statements, and in the rest of this Subsection we will explore some of the implications of this definition. In particular we are interested in an assertion that limiting communication to an upwards direction but allowing closures gives the same expressive power as allowing communication in both directions but not using closures.

First we need a lemma.

Lemma 1.5. *Suppose we have two processors A and B with two wires ab1 and ab2 from A to B. These wires obey the "singly buffered" condition above. It is possible to simulate those two wires with a single wire with no more than a constant factor speed loss.*

Proof: Replace the wire. Replace occurrences of **read**($ab1, x$) (resp. $ab2$) in B with the fragment **while undefined**($vab1$) **do** *check*(); **od**; $x \leftarrow vab1$; $vab1 \leftarrow$ **undefined**. Replace **readable**($ab1$) with **defined**($vab1$). In A, replace **send**($ab1, x$) with **while defined**($vab1$) **do** *check*(); **od**; $vab1 \leftarrow x$; and **sendable**($vab1$) with **undefined**($vab1$).

Insert "*check*()" sufficiently often to guarantee execution periodically, with a period short compared to the time it takes to communicate between processors. The *check*() call in A checks whether $vab1$ and $vab2$ are defined. If either is defined, say $vab1$, *check*() sends the pair $\langle\langle 1, vab1\rangle\rangle$ over the wire and does $vab1 \leftarrow$ **undefined**. The check call in B is a finite state machine. In its initial state it checks whether there is anything to read on the wire; if there is, it reads it. This should be a number i; the FSM enters a state S_i. If *check*() is in S_i, then it will check whether $vabi$ is empty and only if so it will read the next object from the wire and enter the initial state.

Enumeration of the sequences of actions on the two wires, actual and simulated, serve to establish correctness. That there is only a constant-factor slowdown can be derived from the fact that *check*() does a constant amount of work unless it waits, that it only waits if (and as long as) the simulated machine would have waited, and that it replaces each communication with a constant number (two) of communications. ∎

Now we can prove a fundamental theorem about unidirectional communication in a tree.

Theorem 1.8. *Suppose we have a tree parallel structure $\mathcal{T}$ without transmission of closures. Then it is possible to perform the same computation that $\mathcal{T}$ performs on an upward tree parallel structure.*

Proof: Simulate a second wire from each child to its parent per the previous theorem. Call that wire C_l (C_r) where it impinges on the parent and C_p where it impinges on the child.

The nodes' programs must be modified as follows: All parts of the program must remain unchanged except for downward communications, which consist of sending statements of the form (1) **write**(**left**, x) and (2) **sendable**(**left**) (or **right**, of course), and receiving statements of the form (3) **read**(**parent**, x) and (4) **readable**(**parent**). These four forms are directly translated as follows: (1)=**read**(C_l, C); $C(x)$, (2)=**readable**(C_l), (3)=**while undefined**(v)**do** *check*(); **od**; $x \leftarrow v$ $v \leftarrow$ **undefined**; **send**($C_p, \lambda_x^v[v \leftarrow x]$) and **defined**($v$). *check*() is from the previous theorem.

Additionally prepend "**send**($C_p, \lambda_x^v[v \leftarrow x]$)" to former recipients' programs and append "**read**(C_l, C); $C(x)$ to former senders'.

That this causes correct information to be seen in the recipient is evident from the observation that each closure is used to send exactly one value to the recipient, exactly that value is used as an argument to the closure as was previously being sent, and it is only used once (and immediately rendered undefined). That this causes the programs to "hang" at exactly the right times can be easily seen from the fact that there is a closure in (say) C_l exactly when the recipient would have been receptive, and there is a value in v exactly when there would have been a value available. ∎

The key point to note is that all downward communication is expressed as closure application. This suggests that it will be possible to express a problem that apparently can not be solved by divide & conquer as the corresponding problem of creating, in the root, a closure that has a desired result when applied.

We have therefore shown that we do not surrender any expressive power when we limit tree declarations to upward communication.

Chapter 2

Examples of the Use of Closures

§2.1 The Handshake Problem

Suppose we have a pipeline of information. Data are supplied at one end of a chain of processors, processed by every intermediate processor (perhaps in combination with data flowing the other way), and the results are either extracted at the other end or developed in some of the intermediate processors. An example of a problem that can be easily solved with a parallel structure of this sort is convolution, where the specification $\forall \vec{A}, B \, \exists \vec{A}' [\forall i [a'_i = \sum_{j+k=i} a_j b_k]]$ must be met. This can be accomplished by a row of processors, each responsible for computing one element of $\vec{a}'$, and a regimen in which the $\vec{A}$-values flow one way, a_1 first, and the B-values flow the other way, b_n first.

To perform the synchronization using closures, we would have to state that the I/O processors at each end provide a closure that can be used to obtain the next datum.

There are two possible ways that use of a closure can result in data coming to be available to the point of use. Either the value can be returned as the result of the application, or the application can cause the datum to be sent separately by the closure's host.

We prefer the latter. We like closures not to return values, as we would have to invent a syntax to allow other computations to proceed while awaiting an answer. Expressive power is not lost in forbidding closures to return a value, because one can instead have the value returned as a separate communication. Our prime purpose in setting up parallel structures is to allow different processors to do different but related work simultaneously, and this would be compromised by this restriction. I will assume this convention in what follows. It is clear that the difference is one of convenience and not a fundamental one. It does, however, allow for such features as a natural method for having a value be the result of the application of more than one closure.

Frequently a link will be used for more than one value. If it is convenient to use a closure to get succeeding values, there will be many applications of closures. There are two ways to manage this: either a single closure can be invoked several times, or use of a closure can cause a new closure as well as the next value to be sent.

The obvious apparent disadvantage of the latter, that it would seem to require the transmission of extra, useless data, is not real. When the use of a closure causes it to become dead but causes another one to be sent, we have the situation where only one instance of a given class of closure can be live at one time. In this situation the closure need not be sent

If a closure is invoked repeatedly, this causes a problem in determining when a closure is dead. This problem is not unique to this circumstance, however; there can arise a case in which it is not known whether a closure will be used even once.

§2.2 Divide-and-Conquer with Closures

In this Section we will consider the broadcast problem, the prefix summation problem, and a part of one solution to the connected components problem that is amenable to tree solution.

§§2.2.1 Broadcast

In the broadcast problem, a value or values known in a central location are distributed to many locations. The broadcast problem can be described formally as $\forall i[a'_i \leftarrow F(a_i, x)]$ or perhaps $\forall j[\forall i[a'_{ij} \leftarrow F(a_{ij}, x_j)]]$. One method of synthesising solutions to this problem might be to recognize it as a distinct pattern and carry a synthesis rule that produces a broadcast tree when supplied an instance of a broadcast problem. Another solution is to produce a chain of processors as a bucket brigade to distribute the information, and then to successively split the chain in half, but this has the problem that the synthesis process is iterated a variable number of times. With the new mechanism of closure passing, it is possible to provide more general rules that handle broadcast problems as a special case without multiple reformulations.

Consider the application of divide & conquer. We want to produce a closure that, when applied to x_j, performs $\forall i[a'_{ij} \leftarrow F(a_{ij}, x_j)]$. We hypothesise that to solve the problem we for a whole subarray we can solve the problem for each of two pieces of the subarray and combine the two solutions in some manner. Giving the names fl and fr to the closures for the left and right halves of the problem and fw to that for solving the whole problem, we then show that to combine closures $fl{=}\lambda_x[\forall j \in r_1[a'_j \leftarrow F(a_j, x)]]$ and $fr{=}\lambda_x[\forall j \in r_2[a'_j \leftarrow F(a_j, x)]]$ we have only to create $fw{=}\lambda_y^{fl,fr}[fl(y) \parallel fr(y)]$. We go through the following sequence:

$$\begin{aligned}
& \forall A, x\, \exists A'[\forall i \in [1 \ldots n][a'_i = F(a_i, x)]] && \\
& \Rightarrow \exists C(x)[\mathbf{action}(C(x)) = \forall A, x[\forall i \in [1 \ldots n][a'_i = F(a_i, x)]]] && \text{(abstraction)} \\
\text{hypothesis:} \quad & \equiv \; \exists C_l^u[\mathbf{action}(C_l^u(x)) = \forall A, x[\forall i \in [l \ldots u][a'_i = F(a_i, x)] && \text{(division)} \\
& \quad \wedge\, C_l^u(x) \equiv G(C_l^{u'}(G_l(x)), C_{u'+1}^u(G_r(x)))]] &&
\end{aligned}$$

The abstraction step is the step of asserting that there is a function whose application brings about the FOL expression that is being abstracted. The division step is the step of asserting that it is possible to build a closure that solves a large problem, given closures that solve subproblems (and possibly other data).

This can be satisfied by setting $G(C_1(x), C_2(y)) \equiv C_1(x) \parallel C_2(y)$ (concurrent composition) and $G_l(x) = G_r(x) = x$.

It only remains to describe the procedure for handling a singleton array. This is the closure $\lambda_x^i[a'_i \leftarrow F(a_i, x)]$.

The computation of the top level closure is $O(\log n)$ where n is the size of the problem. This is clear from the reasoning of Section 1.2and from the observation that time(G)=$O(1)$. (G is creation of a closure enclosing two given closures.) Similarly, the time consumed by an application of the top closure will be $O(\log n)$ from the fact that max(time(G_l), time(G_r))=$O(1)$. (G_l and G_r are identity operations.)

§§2.2.2 Parallel Prefix

2.2.2.1 Overview

To use the closure technique on a given specification, reformulate the problem from something like $\forall X, \ldots \exists Y[P(X,Y)]$ to $\exists C[\forall X, \ldots \text{action}C() = P(X,Y)]$. Heuristically, the problem is reformulated from that of satisfying a specific input/output specification to that of producing a closure that, when applied, will cause the I/O specification[1] to be satisfied.

We will need to define "augmented prefix summation with augend z" as $\forall 1 \leq i \leq n[a'_i = z + \sum_{1 \leq j \leq i} a_j]$. We then say that the task is to deliver to the root of the tree a closure that will perform augmented prefix summation. To create a closure that will perform augmented prefix summation with augend z on a non-trivial vector, divide it into two halves, get such a closure from each half together with the grand total of the input values for that half, invoke the left half's closure with z as an augend and the right half's with $z +$ the left half's sum. We deliver to each node of the tree closures that will perform augmented prefix summation on the vector comprising its leaves, together with the leaves' sum. Note that the closure delivered to each node's parent has to include the left subtree's sum, which is available now but won't be later. A more formal description follows.

Assume that a vector $A_{[l \ldots u]}$ is divided into $A_{[l \ldots u']}$ and $A_{[u'+1 \ldots u]}$. Further assume that we are trying to compute $F(A_{[l \ldots u]})$ which we will denote F_l^u. Further assume that we want to have some effects, local to the array elements. We would therefore want to compute a closure, C_l^u, that would have the desired effect.

The generic combination operator for the values is $F_l^u = G(F_l^{u'}, F_{u'+1}^u, l, u, u')$ and it is a synthesis task to derive the properties of G. Similarly, $C_l^u = G(C_l^{u'}, C_{u'+1}^u, l, u, u')$. If the closure has an argument the situation is slightly more complex; we have $C_l^u(z) = G(C_l^{u'}(G_l(z, F_l^{u'}, F_{u'+1}^u, l, u, u'), C_{u'+1}^u(z, F_l^{u'}, F_{u'+1}^u, l, u, u'), l, u, u')$ where the F vectors are the values available to (and incorporated in) C_l^u. This general schema need only be used with specific combiners (i.e., G, G_l, etc.). As a simple example, prefix summation can be performed by this schema if $G \equiv (C_{left} \parallel C_{right})$ (where $\parallel$ is concurrent application), $G_l(z) = z$, and $G_r(z) = z + v_l$. v, in turn,is computed as $v_l + v_r$. Singleton v- and C-expressions are $C_i \equiv \lambda z[a'_i \leftarrow a_i + z]$ and $v_i = a_i$.

2.2.2.2 Derivation

In this problem, the specification to meet is $\forall i \in [1 \ldots n][a'_i \leftarrow \sum_{j \in [1 \ldots i]}[a_j]]$. I will introduce the abbreviation $\sum_l^u \equiv \sum_{j \in [l \ldots u]} a_j$. This then becomes $\forall i \in [1 \ldots n][a'_i \leftarrow \sum_1^i]$. We change the

[1] More precisely, the problem of satisfying the I/O specification that requires no input and produces that closure

specification to one requiring the computation of a closure which, when applied to no arguments, performs this action; together with the application of that closure.

$$\forall \vec{A}\, \exists \vec{A}' [\forall i \in [1 \ldots n][a'_i = \sum_{j \in [1 \ldots n]} a_j]]$$

$$\Rightarrow \exists C\, \forall \vec{A}[\mathbf{action}(C()) = [\forall i \in [1 \ldots n][a'_i = \sum_{j \in [1 \ldots n]} a_j]] \qquad \text{(abstraction)}$$

$$\text{hypothesis:} \equiv \exists C^u_l\, \forall \vec{A}[\mathbf{action}(C^u_l()) = \forall i \in [l \ldots u][a'_i = \sum_{j \in [1 \ldots n]} a_j] \qquad \text{(division)}$$

$$\wedge\, C^u_l \equiv G(C^{u'}_l(), C^u_{u'+1}())]$$

But $\mathbf{action}(C^u_{u'+1}()) = \forall i \in [u'+1 \ldots u][a'_i = \sum^i_{u'+1}]$ so this is impossible. $C^u_{u'+1}$ must be provided with a parameter to be able to do this.

We modify the closures so instead of $\mathbf{action}(C^u_l()) \equiv \ldots$ we have $\mathbf{action}(C^u_l(z)) = \forall i \in [l \ldots u][a'_i = H(\sum^i_l, z, i)]$. We do not yet know the properties of H.

We now have:

$$\mathbf{action}(C^u_l(z)) = \forall i \in [l \ldots u][a'_i = H(\sum^i_l, z, i)]$$

$$\mathbf{action}(C^{u'}_l(z)) = \forall i \in [l \ldots u'][a'_i = H(\sum^i_l, z, i)]$$

$$\mathbf{action}(C^u_{u'+1}(z)) = \forall i \in [u'+1 \ldots u][a'_i = H(\sum^i_{u'+1}, z, i)]$$

So we observe

$$\mathbf{action}(C^u_l(z))$$

$$\equiv (\forall i \in [l \ldots u][a'_i = H(\sum^i_l, z, i)]) \qquad \text{(above)}$$

$$\equiv \mathbf{action}(G(C^{u'}_l(G_l(z)), C^u_{u'+1}(G_r(z)))) \qquad \text{(D\&C)}$$

$$\equiv G((\forall i \in [l \ldots u'][a'_i = H(\sum^i_l i, G_l(z), i)]), (\forall i \in [u'+1 \ldots u][a'_i = H(\sum^i_{u'+1}, G_r(z), i)]))(z) \text{(expansion)}$$

$$\equiv \forall i \in [l \ldots u'][a'_i = H(\sum^i_l, z, i)] \wedge \forall i \in [u'+1 \ldots u][a'_i = H(\sum^i_l, z, i)] \qquad (\forall \text{ identity})$$

Assuming G merely generates a closure to produce application of both of its parameters, then $H(\sum^i_l, z, i) = H(\sum^i_l, G_l(z), i)$ and $H(\sum^i_l, z, i) = H(\sum^i_{u'+1}, G_r(z), i)$. The first unifies to $z = G_l(z)$.

The second needs a bit more attention. If we represent $\sum_l^i$ as $\sum_l^{u'} + \sum_{u'+1}^i$, we learn that $H(\sum_l^i, z, i) = H(\sum_l^{u'} + \sum_{u'+1}^i, z, i) = H(\sum_{u'+1}^i, G_r(z), i)$, so $H(q+r, z, i) = H(q, G_r(z), i)$ where $r = \sum_l^{u'}$.

Letting $H = \lambda_{x,y}[x+y]$ we get $G_r = \lambda_x[x+r]$. This leads to another problem, that there isn't enough information around to compute G_r. We have to expand the problem again to bring about the availability of intermediate values for the intermediate closures. In this case we need $\sum_l^{u'}$. Instead of

$$\mathbf{action}(C_l^u(z)) \equiv \mathbf{action}(G(C_l^{u'}(G_l(z)), C_{u'+1}^u(G_r(z))))$$

we want

$$v_l^u = H(v_l^{u'}, v_{u'+1}^u)$$

and

$$\mathbf{action}(C_l^u(z)) \equiv \mathbf{action}(G(C_l^{u'}(G_l(v_l^{u'}, v_{u'+1}^u, z)), C_{u'+1}^u(G_r(v_l^{u'}, v_{u'+1}^u, z))))$$

.

Taking a more intuitive view for the moment, we observe that we want to compute a two-tuple $\langle\langle v_l^u, C_l^u \rangle\rangle$ in which $v_l^u = \sum_l^u$ and in which $\mathbf{action}(C_l^u(z))$ is the computation of an *augmented prefix summation*, where $a_i' \leftarrow z + \sum_l^i$ instead of $a_i' \leftarrow \sum_l^i$.

We want $G_r(v_l^{u'}, v_{u'+1}^u, z) = z + \sum_l^{u'}$, so we must use $v_l^{u'} = \sum_l^{u'}$ or $v_l^u = \sum_l^u$.

We lack only one step to a complete solution. Initially we wanted to compute a closure which, when computed for that "sub-array" which is the whole array and applied to no argument, computes the prefix sum. We will get, instead, a pair of results. One of the results is a value, and the other is a closure which, when applied to *one* value, computes a generalization of the prefix sum. It remains to convert this back into a closure that can be applied to no arguments.

We have

$$\mathbf{action}(C_l^u(z)) = \forall i \in [l \ldots u][a_i' = \sum_j^u + z]$$

and we want

$$\mathbf{action}(F'()) \, \forall i \in [1 \ldots n][a_i' = \sum_1^n] \equiv \mathbf{action}(C_1^n(z)) = \forall i \in [1 \ldots n][a_i' = \sum_1^n + z]$$

for some z. Clearly $z=0$ works.

Summarizing, we have all of the following:

$$\begin{aligned}
\textbf{action}(C()) = \quad & \forall 1 \le i \le n[a'_i = \sum_1^i] \\
\textbf{action}(C_l^u()) = \quad & \forall l \le i \le u[a'_i = \sum_l^i] \\
\equiv \quad & \forall l \le i \le u'[a' = \sum_l^i] \land \forall u'+1 \le i \le u[a' = \sum_l^i] \\
\equiv \quad & \forall l \le i \le u'[a' = \sum_l^i] \\
& \land \forall u'+1 \le i \le u[a'' = \sum_{u'}^i] \\
& \land \forall u'+1 \le i \le u[a' = \sum_l^{u'} + a'']
\end{aligned}$$

We must supply a new parameter:

$$\textbf{action}(C(z_0)) = \forall 1 \le i \le n[a' = H(\sum_1^n, z_0)]$$
$$\textbf{action}(C_l^u(z)) \equiv \textbf{action}(C_l^{u'}(G_l(z)), C_{u'+1}^u(G_r(z)))$$

$H(\sum_l^i, z, i) = H(\sum_l^{u'} + \sum_{u'+1}^i, z, i) = H(\sum_{u'+1}^i, G_r(z), i)$, which works if $H(x,y) = x + y$ and $G_r(z) = \sum_l^{u'} + z$, but the latter requires having $\sum_l^{u'} + z$ available. We therefore further modify the problem by requiring the collection of another value.

$$\begin{aligned}
v_l^u &= \sum_l^u \\
&= H(v_l^{u'}, v_{u'+1}^u) \\
&= H(\sum_l^{u'}, \sum_{u'+1}^u)
\end{aligned}$$

The last observations we need (the base case) are:

$$v_i^i = \sum_i^i = a_i$$

$$C_i^i = \lambda_z[\forall i \le j \le i[a'_j = z + \sum_i^j]] = \lambda_z[a'_i = z + a_i]$$

We therefore have $H(x,y) = x + y$ making $v_l^u = v_l^{u'} + v_{u'+1}^u$. $C_l^u(z)$ applies $C_l^{u'}$ to z, and $C_{u'+1}^u$ to $z + v_l^{u'}$. Creating new symbols for the values (v_l, v_r and v) and closures (C_l, C_r, and C) received from the subproblems and passed to the superproblem, we finally get the following:

$$\begin{aligned}
H(x,y) &\equiv x+y \\
v &= v_l + v_r \\
v.leaf_i &= a_i \\
G(C_l,C_r) &\equiv C_l(G_l(x)) \parallel C_r(G_r(x)) \\
G_l(x) &\equiv x \\
G_r(x) &\equiv x + v_l \\
C &= \lambda_x^{C_l,C_r,v_l}[G(C_l,C_r)] \\
C.leaf_i &= \lambda_x[a'_i = x + a_i] \\
C.root &= \lambda_{()}^{C_l,C_r,v_l}[G(C_l,C_r)(0)]
\end{aligned}$$

This can be converted to a decorated tree structure by simple rewrite rules.

For example, we have $G(C_l,C_r) \equiv C_l(G_l(x)) \parallel C_r(G_r(x))$. We would therefore have a synthesized TREE declaration to read, in part,

```
        .
    inter HAS $C$, $v$
        HEARS *leftson*  (USES $C$ as $C_l$, USES $v$ as $v_l$)
        HEARS *rightson* (USES $C$ as $C_r$, USES $v$ as $v_r$)
        TALKS *parent*   (SENDS $C$, SENDS $v$)
        .
```

and the program for the internal nodes to read, in part,

```
(in T.inter):
  $C \leftarrow \lambda_x^{v_l,v_r,C_l,C_r}[C_l(G_l(x)) \parallel C_r(G_r(x))]$
    where $G_l(x)=x$
    where $G_r(x)=x + v_l$
  $v \leftarrow v_l + v_r$
```

§§2.2.3 Connected Components

The problem is to find the connected components of a graph, given an adjacency matrix (a matrix A in which a_{ij}=true iff node i is (directly) connected to node j in the graph. The adjacency matrix will be available for input one row at a time, and a solution is better that reads the rows at constant intervals.

In this Subsection we will derive a tree structure that solves part of the problem and meets certain worst case time constraints. The derived structure will operate while the rows of the adjacency matrix are read in.

Formally, we will assume that there exists a source of rows of the adjacency matrix that can provide one row at a time. Each column will be read by its own processor. Columns and rows have integers in the range $[1,2,\ldots,n]$ as names. When column i's processor reads row j it receives the value true if there is a graph edge between i and j or false otherwise. The network we derive will then store the information in such a manner that it or some other network can identify connected components of the graph whose adjacency matrix was read. The identification process is not the issue here.

The column processor nodes of the network must read elements of the rows of the adjacency matrix at such a time (in relation to the time other processors read their elements of the same row) that the network will not confuse elements of different rows of the matrix, and the net must build a representation of the the (partial) connected components information in some useful manner. The representation should be compact and the computation should be fast.

First we will derive the structure up to one important implementation decision; then we will describe the two resulting parallel structures.

2.2.3.1 Derivation of a Tree Structure

In the connected components problem, we do not necessarily want to change the state of the leaves of the tree or develop a value at the root. Instead, we want to change some state so questions about connected components become easier to answer.

We will use the notation $CC(i)$ to denote the set of nodes in the same connected component as the node i. $CC'(N)$ is a predicate indicating whether all nodes of N, a set of nodes, are in a single connected component. Since the state of knowledge of the connected components of a graph can vary with time and, in a multiprocessor system, with location, we will later introduce other variants of the CC' predicate.

We will read the rows of the adjacency matrix one by one. After we have read all of the rows we will then engage in another computation, not described here, to put $\mathbf{reduce}_{\min}\{j : j \in CC(i)\}$ in leaf i. In what follows we will call the processing that takes place between the reading of consecutive rows of the matrix a *phase*.

There are several solutions to the connected components problem which we reject because they have certain undesirable features. One solution, for example, would be to have each node record the row numbers of all rows of the adjacency matrix in which it is mentioned. This would require a lot of storage. Another solution is to have each leaf, after each row, find $\mathbf{reduce}_{\min}\{j : j \in CC(i)\}$ so far. This solution has the problem that the time between the reading of rows can vary over a wide range.

Our derivation requires a certain amount of invention. We will assume that the user provides this by defining several intermediate predicates and by providing some information. First, the idea of a map to store the state of the connected components so far, and than the idea that the map is limited, have to be conceived.

We start with axioms about connected components:

$$CC'(\{e\})$$

$$CC'(\emptyset)$$

$$CC'(A) \wedge CC'(B) \wedge A \bigcap B \neq \emptyset \Rightarrow CC'(A \bigcup B)$$

$$CC'(A) \wedge A' \subseteq A \Rightarrow CC'(A')$$

We observe that the following is trivially true:

$$CC'(A) \wedge CC'(B) \wedge \exists a,b[a \in A \wedge b \in B \wedge CC'(\{a,b\})] \Rightarrow CC'(A \bigcup B)$$

First, we supply TRANSCONS with a divide-and-conquer formulation.

```
$\forall V, W \in V CC'(W)$
  where
$CC'(W) \equiv |W| \leq 1$
    $\vee$
        $W = W_l \uplus W_r$
      $\wedge$ $CC'(W_l)$
      $\wedge$ $CC'(W_r)$
      $\wedge$ $(W_l \neq \emptyset \wedge W_r \neq \emptyset \Rightarrow CC'(\{\text{arb } W_l, \text{arb } W_r\}))$
```

TRANSCONS can easily check that this meets the axioms, but the combination of the two halves by a pair of arbitrary elements, one from each half, constitutes a user-supplied invention.

TRANSCONS observed that the current state of CC' is represented by the choices of pairs of arbitrary elements, and introduces M to carry this information. Since M represents the state of knowledge of connected components, we will define a new binary predicate $CC(M,X)$ which denotes that the mapping M asserts that there exists a connected component C such that $X \subseteq C$. Taking a finite difference against the addition of a new set X that is known to be connected, we get:

```
$\forall X, M \exists M'[CC(M',X) \wedge \forall W[CC(M,W) \Rightarrow CC(M',W)]$
            $\wedge \forall a,b[\sim CC(M,\{a,b\})$
                    $\wedge \forall Y,Z[CC(M,\{a\} \bigcup Y) \wedge CC(M,\{b\} \bigcup Z)$
                        $\Rightarrow Y \bigcap X = \emptyset \vee Z \bigcap X = \emptyset]$
                      $\Rightarrow \sim CC(M',\{a,b\})]]$
  where
$CC(M,W) \equiv |W| \leq 1$
    $\vee$
        $W = W_l \uplus W_r$
      $\wedge$ $CC(M,W_l)$
      $\wedge$ $CC(M,W_r)$
      $\wedge$ $(W_l \neq \emptyset \wedge W_r \neq \emptyset$
            $\Rightarrow \exists a \in W_l, b \in W_r[M(a,b)])$
```

The long conjunct on the second through fifth lines state simply that no connected components are implied by M' that aren't either implied by M or forced by X.

We invite the user to make another critical observation, namely that $\forall W[CC(M,W) \Rightarrow CC(M',W)]$ can be satisfied by $\forall a,b[M(a,b) \Rightarrow M'(a,b)]$. (S)he can further observe from the original axioms that $CC(\{a,b\} \wedge a \in A \wedge CC(\{b,c\}) \wedge c \in C \Rightarrow CC(A \bigcup C)$. We can thus liberalize the condition on M in CC as follows:

```
$\forall X, M \exists M'[CC(M',X) \wedge M(a,b) \Rightarrow M'(a,b) \wedge \ldots]$
  where
$CC(M,W) \equiv |W| \leq 1$
```

$$
\begin{array}{l}
\lor \\
\quad W = W_l \uplus W_r \\
\quad \land \; CC(M, W_l) \\
\quad \land \; CC(M, W_r) \\
\quad \land \; (W_l \neq \emptyset \;\land\; W_r \neq \emptyset \\
\qquad \Rightarrow \exists a \in W_l, b[M(a,b) \land (b \in W_r \lor CC(M, \{b\} \cup W_r))])
\end{array}
$$

This specification is suboptimal because it allows M to be multivalued. We will examine this solution in detail and see how it translates into decorated trees that maintain M in internal state. We will then see what can be done to improve this.

We therefore make a change in CC to express the fact that the divisions will always be made in the same manner, and that M need only be defined for one set of subsets of the universe. This change is the addition of a parameter, a subset of the universe (of nodes in the graph whose connected components we are seeking). Later we will repair another deficiency of this specification, that it allows M to be larger than we would like.

M will be made a ternary rather than a binary relation. $M(S,a,b)$ is true if a connects to b relative to S. The purpose of this is to limit the size of M.

A new parameter to CC ranges over particular subsets of the universe. It has two roles: it tells what version of M to use, and it restricts acceptable solutions to CC. $CC(S,M,X)$ is true only if there exist elements of $M(S',x,y)$, where $S' \subseteq S$, that show that X is connected. This is a stronger condition than the original $CC(M,W)$.

To formalize the new parameter of CC we write:

$$
\begin{array}{l}
\forall V, X, M, W \in V \,\exists M' \forall a, b[CC(M,W) \land CC(M',X) \land M(S,a,b) \Rightarrow M'(S,a,b)] \\
\quad \text{where} \\
CC(M,W) \;\equiv\; CC(U,M,W) \\
\quad \text{and} \\
CC(S,M,W) \;\equiv\; |W| \leq 1 \\
\quad \lor \\
\qquad W_l = W \cap L(S) \\
\qquad \land \; W_r = W \cap R(S) \\
\qquad \land \; CC(L(S), M, W_l) \\
\qquad \land \; CC(R(S), M, W_r) \\
\qquad \land \; (W_l \neq \emptyset \;\land\; W_r \neq \emptyset \\
\qquad\quad \Rightarrow \exists a \in W_l, b \in W_r[M(S,a,b)]) \\
\quad \text{and} \\
L(S) \uplus R(S) = S
\end{array}
$$

Now we can perform a synthesis by transforming **satisfy**($\forall V, X, M, W \in V \,\exists M' \forall a, b[CC(M,W) \land CC(M',X) \land M(S,a,b) \Rightarrow M'(S,a,b)]$). This works with no problems. We soon find ourselves transforming **satisfy**($M'(S,a',b')$). However, this causes no problem. Yet.

Suppose we add an additional condition, $M(S,a,b) \land M(S,a,c) \Rightarrow b=c$. We start with this: (we have replaced occurrences of M by occurrences of M', as the constraint propagator would do when analyzing "$CC(S,M',W)$".)

$$\wedge \ (W_l \neq \emptyset \ \wedge \ W_r \neq \emptyset$$
$$\Rightarrow \exists a \in W_l, b \in W_r[M'(S,a,b) \ \wedge \ \forall c[M'(S,a,c) \Rightarrow c=b]])$$

. .

This last clause makes us a bit unhappy, when considered together with the expression $M(S,a,b) \Rightarrow M'(S,a,b)$

However, we have $M(S,a,c) \Rightarrow CC(S,\{a,c\})$ and $CC(R(S),\{c\} \bigcup W_r) \ \wedge \ CC(S,\{a,c\}) \Rightarrow CC(S,\{a\} \bigcup W_r)$.

We therefore use $\vee$ to expose the fact that there are alternatives:

. .

$$\wedge \ (W_l \neq \emptyset \ \wedge \ W_r \neq \emptyset$$
$$\Rightarrow \exists a \in W_l, b \in W_r[(M'(S,a,b) \ \wedge \ \not\exists c \neq b[M(S,a,c)]$$
$$\vee \ \exists c[M'(S,a,c) \ \wedge \ CC(S,\{c\} \bigcup W_r)]])])$$

. . .

As it is known that $M(S,a,x)$ can only be asserted by the above, an inductive proof is available that $c \in R(S)$. This can therefore be replaced by

. .

$$\wedge \ (W_l \neq \emptyset \ \wedge \ W_r \neq \emptyset$$
$$\Rightarrow \exists a \in W_l, b \in W_r[(M'(S,a,b) \ \wedge \ \not\exists c \neq b[M(S,a,c)]$$
$$\vee \ \exists c[M'(S,a,c) \ \wedge \ CC(R(S),\{c\} \bigcup W_r)]])])$$

. . .

This gives two alternative ways to **satisfy** the specification. We can **satisfy** $M'(S,a,b)$ if $M(S,a,b) \vee \not\exists_c[M(S,a,c)]$. **satisfying** the other disjunct is harder than this because it requires satisfaction of a predicate containing $R(S)$, so we prefer the first disjunct when it can be **satisfyed**. If we can't use the first disjunct, then we know $\exists c[M(S,a,c)]$ so we have only to **satisfy** $CC(R(S),\{c\} \bigcup W_r)$ for that c. This leads to:

```
satisfy(∃ $a \in W_l, b \in W_r$
        $[(M'(S,a,b) \wedge \not\exists c \neq b[M(S,a,c)] \vee \exists c[M'(S,a,c) \wedge CC(R(S),\{c\} \bigcup W_r)]])])$
   →    bind $a$ to arb($W_l$), $b$ to arb($W_r$) in
      if $M'(S,a,b) \vee \not\exists_c[M(S,a,c)]$ then satisfy($M'(S,a,b)$)
            else satisfy($M(S,a,c) \Rightarrow CC(R(S),\{c\} \bigcup W_r)$)
```

2.2.3.2 Alternative Data Structures

It is now necessary to consider the options for storing M. The type of M is $T \times U \rightarrow U$, where U is the set of nodes in the graph whose connected components are being determined, and T is a set of sets such that $U \in T \wedge (S \in T \wedge |S| > 1 \Rightarrow R(S) \in T \wedge L(S) \in T)$. The genesis of T is such that each intermediate node plus the root of the tree has as its set of leaves some element of T if each element of U is represented by a leaf.

Because of the type of M, we have four simple options to represent the mapping: We can represent it in one processor's memory, in the memory of one processor per element of T, in one processor per element of U, or in one processor per element of $T \times U$. The first possibility would lack concurrency and the last would require too many processors. The remaining possibilities include using interior nodes of the tree (corresponding to elements of T) or leaves (corresponding to elements of U) as the repository for information about parts of M.

Inspection of the specification yields the information that the tree node representing a set S must be able to answer questions of the form $\exists c[M(S,a,c) \wedge c \neq b]$ and find c suchthat $M(S,a,c)$, and must be able to **satisfy**$(M(S,a,b))$. This requires either keeping $M(S,x,y)$ in S's node or providing that node with appropriate closures.

That node must also be able to **satisfy**$(CC(L(S),M',W_l))$ to **satisfy**$(CC(R(S),M',W_r))$, and to **satisfy**$(CC(R(S),M',c \bigcup W_r))$ given $c \in R(S) \wedge CC(R(S),M',W_r)$. This requires another handful of closures.

Since closures to **satisfy**$(CC(L(S),M',W_l))$ and **satisfy**$(CC(R(S),M',W_r))$ would require only information available below $L(S)$ and $R(S)$ respectively, and since there is no control flow path by which the need to satisfy these two predicates would be evaded, we observe that each interior node requires a=**arb** W_l, b=**arb** W_r, and the closure $\lambda_x^{M',a,b}[\textbf{satisfy}(M'(R(S),a,x))]$.

We are building a map that maps at most one leaf of the right subtree to each leaf of the left subtree. As described, the map is stored in the node that has the appropriate subtrees. However, other alternatives are possible.

There are three natural places to store the assertion $M(S,a,b)$. They are the node whose subtree's leaves are S, leaf a and leaf b. If the information is stored in S, there must be one cell for each leaf of the left subtree, and if the information is stored in a then there must be one cell for each ancestor representing S. If the information is stored in b we have no limit (beyond the size of the problem) for the amount of storage that must be provided in b. We therefore reject this alternative.

Storing M in the node heading S minimizes communication (information is where it is used) making the algorithm take $O(\log n)$ steps. These steps are not constant-time steps because they require access to a random access memory whose size is $O(n)$, itself an $O(\log n)$ operation[2]. The algorithm therefore has an $O(\log^2 n)$ running time.

The result could be transformed to place the fact of $M(S,a,b)$ in a. This would result in a different algorithm, one that requires the leaves to supply closures to access and modify the map.

There is an interesting problem here. We would prefer that the leaves not have to know about elements of T. It would therefore be necessary to have the M table within each leaf organised in a certain order and to have use made of this information in that fixed order. This requires that a "flame front" of subtree handling be arranged such that initially the root is the tree for which you are trying to associate pairs of elements, and on succeeding subphases the level at which we are trying to match descends. This algorithm has an $O(\log^2 n)$ execution time because there are $\lg n$ subphases, each of which is $O(\log n)$.

[2] The constant factors are such that this is probably not a serious issue. If the problem instance is large, say $\geq 2^{10}$ or so, the RAM access time might be slow. However, much of the communication between tree's processors would then be off-chip, making interprocessor communication even slower. If the problem instance is small, the RAMs in each processor would be small enough to make their access time comparable to ordinary logical elements in the processor. Only for a truly immense problem instance, say 2^{30}, would the memory access time dominate the communication time.

We prefer the former data structure, in which $M(S,a,b)$ is represented in S, because the issue described in the previous paragraph does not arise. That structure will always be available to us unless the size of a change to M is proportional to the size of S, and this can not be because the combination step of the divide and conquer scheme must be fast for the specification to parallelize well in a tree structure.

2.2.3.3 Results of Storing the Map in the Leaves

This Subsubsection will discuss the algorithm's response to a single row of input.

The parallel structure is (informally) as follows:

There is a balanced binary tree of processors. The leaves of the tree correspond to the nodes of the graph, and they are ordered in the order that corresponds to the arrivals of rows of the adjacency matrix. (This last fact is not important.) For simplicity of exposition we will write the following as if the leaves *were* rather than "corresponded to" the nodes. For simplicity we will assume that the entire adjacency matrix is supplied, rather than only a triangular matrix.

The leaf nodes build approximations to the answer as the algorithm grinds on. Each leaf node has one memory cell for each ancestor. Consider the memory cell for ancestor a in leaf l_i. It is initialized to the distinguished value nil, and during the course of the algorithm it will come to contain some j such that LCA(j,i)=a and i and j are known to be in the same connected component, provided that some such j exists.

The algorithm works as follows: A leaf is called *active* if its bit is set in the current row of the adjacency matrix. After a row is read in, information is passed upward so each node can determine whether both of its subtrees contain active leaves, and what the highest and lowest active leaves are for such nodes. Information is than passed downward so each internal (or root) node can determine whether it is the top such node. That node sends a message to those two extreme nodes informing them of each other's identity.

The following cycle is repeated

TU computes spans, TD distributes span information and keeps track of the topness of nodes.

```
TU istype TREE ($i$), $i \in [1,\ldots,n]$ size $n$
   root HAS minact, maxact, topp, listop, ristop
         HEARS leftson  (uses upmin)
         HEARS rightson (uses upmax)
         TALKS leftson  (sends listop)
         TALKS rightson (sends ristop)
   inter HAS minact, maxact, topp, listop, ristop
         HEARS leftson  (uses upmin)
         HEARS rightson (uses upmax)
         TALKS leftson  (sends listop)
         TALKS rightson (sends ristop)
         TALKS parent   (sends upmin)
                        (sends upmax)
   leaf HAS $active_i$, $ccmate_{ij}, j \in$ ancestors
         HEARS INPUT    (uses $adj_{ij} j \in [1,\ldots,n]$)
         TALKS parent   (sends upmin)
                        (sends upmax)
```

```
(in TU.leaf_i)
$\forall j \in$ ancestors
  $ccmate_{ij} \leftarrow$ nil
$\forall j \in \langle\langle 1, \ldots, n \rangle\rangle$
  $temp \leftarrow a_{ij}$
  $upmin \leftarrow upmax \leftarrow$ if temp then i else nil
  $dmin \leftarrow downmin$
  $dmax \leftarrow downmax$
  $other \leftarrow$ nil
  $pivot \leftarrow pivot$
  if $dmin = i$ then $other \leftarrow dmax$
  if $dmax = i$ then $other \leftarrow dmin$
  if $other \neq$ nil then
    if $ccmate_{i,pivot}$=nil
      then $awaken \leftarrow$ nil; $ccmate_{i,pivot} \leftarrow other$
      else $awaken \leftarrow ccmate_{i,pivot}$

(in TU.inter)
; first establish my status
 $\langle\langle lrangel, lrangeh \rangle\rangle \leftarrow lrange$
 $\langle\langle rrangel, rrangeh \rangle\rangle \leftarrow rrange$
 $range \leftarrow \langle\langle \min(lrangel, lrangeh), \max(rrangel, rrangeh) \rangle\rangle$
 $livep \leftarrow range_1 \wedge range_2$
; This is a once—per—minor—phase activity
 while $dstatus \neq$'dead

(in TU.root)
 $\langle\langle lrangel, lrangeh \rangle\rangle \leftarrow lrange$
 $\langle\langle rrangel, rrangeh \rangle\rangle \leftarrow rrange$
 $range \leftarrow \langle\langle \min(lrangel, lrangeh), \max(rrangel, rrangeh) \rangle\rangle$
 $livep \leftarrow range_1 \wedge range_2$
 while $dstatus \neq$'dead

(in TD.inter)
 if $pstatus \in$ {'live, 'top}
     then $status \leftarrow$'live
          $range \leftarrow prange$
     elseif $livep$ then $status \leftarrow$'top
                    $range \leftarrow range$
     else $status \leftarrow$'dead
 while $status \neq$'dead

(in TD.root)
     if $livep$ then $status \leftarrow$'top
                 $range \leftarrow range$
            else $status \leftarrow$'dead
```

Each minor phase the leaves sent up awakening info and get back a packet of info very similar to the one they got in the beginning.

Each leaf, when it dies (finds out that the node just above it is dead) sends up an "init" message. When every node has done so the rood broadcasts its own form of "init" and the leaves read from the I/O processor that contains the next row of the adjacency matrix.

Here we describe the *overall* behavior of the algorithm, considering the parallel structure to be a single entity that can do things sequentially. To actually have this effect, there are syncaronication problems, and below we describe a nodes' eye view of the situation, including the work that each node has to do to coordinate with its neighbors.

Initialize:	Have each node read in its element of the adjacency matrix. Those nodes reading a "1" in the adjacency matrix turn themselves on, as does the node whose index corresponds to that of the row of the matrix. Mark the root as the "focus".
Survey:	Every leaf sends information telling whether it is awake. Using this information, the internal nodes below a focus find out which of them has awake descendants in each of the two trees ("has two active subtrees"). This is a straightforward "up" problem.
New root:	The highest node with two active subtrees is determined. This is the Least Common Ancestor (LCA) of active leaves. It becomes the new focus, nodes between it and leaves become "active", and nodes above it but below and including the old focus become "dead".
Tournament:	Select an arbitrary active leaf node in each of each focus's two subtrees. Report the identities of the two leaves to their focus. Simultaneously report the identity of the focus and of the other leaf to each of the two leaves.
Lookup:	The leaves contain a variable mapping mapping their ancestors into a leaf index or the distinguished value nil. The leaves look up the focus in this mapping. If it is nil, they store the other leaf's identity. If the left leaf's value is not nil, report the value to its focus.
New awakening:	If its left tree reports a leaf ID per Lookup, a focus sends a message to that leaf commanding it to awaken.
Refocus:	Each focus sends a message to those of its children that are not leaves telling them to become new focuses, and dies.
Repeat (Maybe):	If not all leaves have a dead parent, go back to New Root.

As can be seen above, the algorithm has several subphases, as the focus moves down towards the leaves, and each of these subphases has several sub-sub-phases: Survey, New root, Tournament, Lookup, New Awakening, Refocus, and Repeat (maybe). Internal nodes of the tree have the status dead, focus or live, and leaf nodes either have status awake or asleep. The behavior of each node during each sub-sub-phase will be described.

Survey: Leaves tell parents whether they are active. Intermediate nodes: (live and focus only) Get status from descendants. Remember and (live only) tell parent how many subtrees have one or more active subtrees. Remember which subtree was active if exactly one was.

New root: If a focus has two active subtrees it tells its left (resp. right) child "focus above you==(*node*), you are left (resp. right)". If it has one, tell that one "focus at or below you" and the other "die". It can't have none.

Intermediate nodes below a focus (i.e., those nodes that are live) listen to their parents. If one hears "die" it dies. If one hears "focus above $=xxx\ldots$" it relays the message and becomes or remains live. If one hears "focus at or below" it acts like in the paragraph above.

Leaves that receive a "die" message send their parent an "I died" message and prepare to read the next line of the adjacency matrix.

Active leaf nodes record the name of their focus.

Tournament and Lookup: Each leaf contains a mapping M relating the name of each of its ancestors to either nil or the index of a leaf. A sleeping leaf node sends nil to its parent. An awake leaf node i that receives a "focus above you=(*node*), you are left" message sends to its parent either $\langle\langle$**empty**$, i\rangle\rangle$ if $M(node)=$**nil**, or $\langle\langle$**loaded**$, M(node)\rangle\rangle$. If it receives "focus above you=(*node*), you are right", it sends i to its parent.

A live internal node which receives nil from both children sends the same to its parent. one that receives something else from one child sends that value to its parent, and one that receives non-nil values from both children sends *either* to its parent. The correctness of the algorithm does not depend on this choice, which can be random, pseudo-random, or consistent.

Each focus receives a message from each child. Say the right child's message is j. If the left child's message is $\langle\langle$**empty**$, i\rangle\rangle$, then $\langle\langle$**record**$, focus, i, j\rangle\rangle$ is sent to the left child and nil is sent to the right. If the left child's message is $\langle\langle$**loaded**$, i\rangle\rangle$, then nil is sent to the left child and $\langle\langle$**awaken**$, i\rangle\rangle$ is sent to the right.

Lookup and New Awakening: Internal nodes relay parents' messages to their children.

If leaf node i receives $\langle\langle$**record**$, focus, i, j\rangle\rangle$ it sets $M(focus) \leftarrow j$. If it receives $\langle\langle$**awaken**$, i\rangle\rangle$ it awakens. (If i doesn't match it does nothing.)

Refocus: Each focus sends its children a "become a focus" message and dies. A live node receiving such a message from its parent changes its status to "focus". A leaf receiving such a message form its parents sends the latter an "I died" message.

Repeat (maybe): At all times, a node receiving two "I died" messages sends one upward. If a node receives a "become a focus" message it sends its children a "begin survey" message. Live intermediate nodes relay such a message, and leaf nodes receiving a "begin survey" message proceed as in Survey.

2.2.3.4 Results of Storing the Map in Internal Nodes

The tree-structured algorithm of 2.2.2.3 uses $O(\log^2 n)$ time per row of the adjacency matrix. More importantly, this constant factor includes a communication between adjacent nodes. It is impossible to do better assuming that the information required to reconstruct connected components is to be kept in the leaves and that there is only to be a logarithmic amount of information in each leaf. The reason for this is that the action taken by the right subtree of a given node depends on information present only in the left subtree, and that the right subtree's recursive analysis of its pattern of leaves to be linked can, in turn, depend on the results of this feedback. We therefore have a logarithmic number of steps, each of which takes $O(\log n)$ time.

It is possible to reduce the constant factor, but only by distributing the information differently.

Instead of having a cell in each leaf for each of its ancestors, suppose we have a cell in each ancestor for each of its leaves. The same number of cells are required, one for each leaf/ancestor pair[3]. Each internal node contains a map which maps names of leaves of the left subtree into either nil or names of leaves of the right subtree.

The overall view of the algorithm is as follows:

Each leaf sends its parent its name if its active, or nil. Each intermediate or root nodes sends its parent either the name of any active node it receives from its children, or nil if it receives nil from both children. If it receives two names it chooses arbitrarily. Each intermediate node also remembers what it received from its children.

In addition, suppose it receives a name from both children. There are two cases. If the name from the left node maps (in the node's internal mapping from leaves to values) into nil, make it map into the name from the right node and do nothing else. If it maps into (say) i, send awaken i to the right child and do nothing else.

If an intermediate node receives an awaken i node from its parent, it checks to see whether i is in its right or left subtree. It also checks to see what it has received before.

If a node receives an awaken i message and has already received a name from i's subtree it sends awaken i message to the appropriate child. If it hasn't so received it considers itself to have so received. (This can involve reacting to further awaken messages, or it can involve looking up either i (if i belongs in the left subtree) or the previously received name (if i was in the right subtree and the previous name was in the left) in the mapping and either extending the mapping or creating a new awaken message.)

The root sends its children "ok" when it's done. Intermediate nodes relay such "okay" messages. Each leaf reads the next line of the adjacency matrix when it receives this ok, and starts a new cycle.

The "wrapup", where each leaf gets the name of a representative of its connected component, is also faster under this arrangement. The root sends its right child its correspondences one by one, followed by "end". When a node receives $a \rightarrow b$ it replaces $b \rightarrow c$ (if it has one) by $a \rightarrow c$. This is not done for $b \rightarrow$ nil. Intermediate nodes also relay correspondences received from parents. When an intermediate node receives "end" from its parent, it dumps its own correspondences as they now stand and then sends its own "end". A leaf node initializes a cell to its own name and a cell named b changes this value to a if it receives $a \rightarrow b$. A leaf node knows it has the right value when it sees "end".

[3]and it should lay out reasonably nicely because the bigger nodes are closer to the root of the tree

Chapter 3

Use of Additional Techniques – Binary Addition

§3.1 Notation

In what follows, we will assume that a problem instance resides in vectors A and B, each containing individual "bits" a_i resp. b_i for $0 \leq i \leq n-1$. The two states of a bit are represented by the values 0 and 1. This discussion is specialized to binary integers, but any radix can be used by reinterpreting the logical operators as follows: $\otimes \equiv +$ (mod (the radix)), $\wedge \equiv \lambda_{x,y}[x+y \geq \text{(the radix)}]$, $\sim \equiv \lambda_x[\text{(the radix)}-x]$, and $\vee \equiv \lambda_{x,y}[x+y \geq \text{(the radix}-1)]$. We apply logical operators to the values 0 and 1, interpreting 0 as false and 1 as true. A represents $\sum_{0 \leq i \leq n-1} a_i 2^i$ and B likewise. The answer is similarly represented in C. We will have occasion to refer to $carry_i$, the carry coming into position i. We use $\otimes$ as the symbol for "exclusive OR".

Our starting point for all of the syntheses in this paper will be the specification:

```
; we want to add $A+B$ where $A=a_{n-1}\ldots a_1 a_0$ and $B$ similarly.
$\forall\, 0 \leq i \leq n-1$
  $c_i = a_i \otimes b_i \otimes \underset{j<i}{\exists}[a_j \wedge b_j \wedge \underset{j<k<i}{\forall}[a_k \vee b_k]]$
```

Figure 2. Our "Standard" Specification of Binary Addition

A derivation of this specification from the "grade school" specification for addition

```
$carry_0 = 0$
$\forall\, 0 \leq i \leq n-1$
  $c_i = a_i \otimes b_i \otimes carry_i$
  $carry_{i+1} = (carry_i \wedge (a_i \vee b_i)) \vee (a_i \wedge b_i)$
```

Figure 3. "Grade School" Specification for Binary Addition

Figure 3. "Grade School" Specification for Binary Addition

is beyond the scope of this paper, although a derivation of the latter from the former will be briefly sketched.

Frequent reference is made of a system called TRANSCONS. This is the TRANSformational CONcurrency Synthesizer (described elsewhere [King-83] [KingMayr-84]) which we are developing at Kestrel. TRANSCONSis an architecture synthesis system which can be used to transform high level specifications into parallel structures. Its features of interest here include the ability to synthesize tree structured processor networks from specifications, and the ability to reformulate concurrent computations by reorganizing the work differently among a collection of processors.

§3.2 Carry Look-ahead Circuit

Consider the definition of Figure 1. The problem with directly synthesizing solutions to this by the methods of TRANSCONS resides in the nesting of quantifiers such that the bound variable of the outer quantifier is one end of the range of the inner one. The reason this is a problem is that it forces the computation of $\theta(n^2)$ boolean values, namely $\forall_{j<k<i}[a_k \vee b_k]$ for each $0 \leq j < i \leq n-1$ (a total of $n(n-1)/2$ (i,j) pairs).

§§3.2.1 Quantifier Levelling

When the following equivalences are applied successively (brief proofs appear in the Appendix)

$$\underset{l<x<u}{\forall}[P(x)] \equiv \max_{x<u}[\sim P(x)] \leq l \qquad (\forall\text{-}to\text{-}max)$$

$$\underset{x<u}{\exists}[P(x) \wedge F(u) \leq x] \equiv \underset{F(u) \leq x<u}{\exists}[P(x)] \qquad (constraint\text{-}to\text{-}binder)$$

$$\underset{l \leq x<u}{\exists}[P(x)] \equiv \max_{x<u}[P(x)] \geq l \qquad (\exists\text{-}to\text{-}max)$$

we get the following sequence of assignments to c_i (changes underlined):

$$c_i = a_i \otimes b_i \otimes \underset{j<i}{\exists}[a_j \wedge b_j \wedge \underset{j<k<i}{\forall}[a_k \vee b_k]]$$

$$\Rightarrow c_i = a_i \otimes b_i \otimes \underset{j<i}{\exists}[a_j \wedge b_j \wedge \underline{\max_{k<i}[\sim(a_k \vee b_k)] \leq j}]$$

$$\Rightarrow c_i = a_i \otimes b_i \otimes \underset{\underline{\max_{k<i}[\sim(a_k \vee b_k)] \leq j<i}}{\exists}[a_j \wedge b_j]$$

$$\Rightarrow c_i = a_i \otimes b_i \otimes \underline{\max_{j<i}[a_j \wedge b_j] \geq} \max_{k<i}[\sim(a_k \vee b_k)]$$

It should be observed that the first transformation solved the basic problem of the need to compute $\theta(n^2)$ values, and that after the last transformation it is possible to do both enumerations in parallel. The bound variable of one enumeration is no longer an endpoint of the range of the other. This means that where we would previously have had approximately $n^2/2$ data items to consider, we now have approximately $2n$.

We now have

$$\begin{array}{l} \forall\, 0 \leq i \leq n-1 \\ \quad c_i = a_i \otimes b_i \otimes \max_{j<i}[a_j \wedge b_j] \geq \max_{k<i}[\sim(a_k \vee b_k)] \end{array}$$

It is possible to express this as an inequality between corresponding elements of the results of two parallel prefix computations as follows:

$$\begin{array}{ll} \forall\, 0 \leq i \leq n-1 & \\ \quad and_i = a_i \wedge b_i & \\ \quad nor_i = \sim(a_i \vee b_i) & \\ \quad max1and_i = \mathbf{if}\ {}^{a}nd_i\ \mathbf{then}\ i\ \mathbf{else}\ -\infty & \\ \quad max1nor_i = \mathbf{if}\ nor_i\ \mathbf{then}\ i\ \mathbf{else}\ -\infty & \\ \quad maxand_i = \max_{0 \leq j<i}[max1and_j] & * \\ \quad maxnor_i = \max_{0 \leq j<i}[max1nor_j] & * \\ \quad C_i = a_i \otimes b_i \otimes (maxand_i \geq maxnor_i) & \end{array}$$

TRANSCONS will be able to synthesize the usual parallel prefix tree structure [**Browning-80**] for each of the two lines marked by an asterisk above. Most of the details of this synthesis are beyond the scope of this paper, but the tree structure comes from uses of divide-and-conquer. The intermediate steps, taken from [**KingMayr-84**], are shown in the Appendix.

There are two parallel prefix trees in the addition parallel structure; one for the variable named *maxand* and another for *maxnor*. The overall structure is shown below.

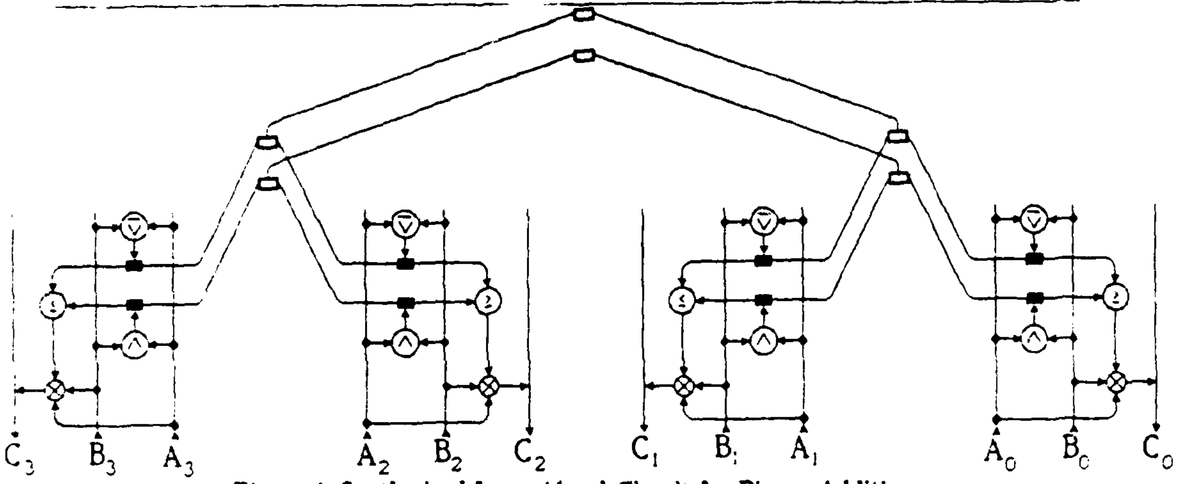


Figure 4. Synthesized Look-Ahead Circuit for Binary Addition

There are two important differences between this structure and standard ones [Hwang-79].

- Because the parallel prefix trees are required to handle integers in the interval $[0, n]$, the size of the nodes and the width of the data paths within the trees are $\theta(\lg(n))$. In the standard network it would be $\theta(1)$. This can be alleviated by some careful reasoning, to be described below.
- Because of the nature of the parallel prefix network synthesized by TRANSCONS, each node is partially responsible for the choreography in its local region. The importance of this fact is that either the nodes need be big enough to participate in an asynchronous data transfer protocol with a handshake, or a global clock must be provided. This is not a serious problem because other parallel prefix networks could have been used (and incorporated into TRANSCONS), and because a three- or five-inverter clock [ConMead-80] can easily be included on the chip if necessary.

§§3.2.2 Data Path Width Reduction

To reduce the width of the data paths and still use a parallel prefix network, an associative operation with constant range and domain must be used.

Now either $\max_{0 \leq j \leq i}[maxland_j] = \max_{0 \leq j \leq i+1}[maxland_j]$ or $\max_{0 \leq j \leq i+1}[maxland_j] = i + 1$, and similarly for *maxlnor*. A case analysis could show that we would have the following table:

$maxand_i \geq maxnor_i \Rightarrow$ $\Downarrow and_{i+1}\ nor_{i+1} \Downarrow$	true	false
and	true	true
nor	false	false
*both**	true	true
neither	true	false

(*this is impossible but knowledge of this fact is unnecessary for the argument)

The effect of and_{i+1}, nor_{i+1}, and_{i+2} and nor_{i+2} on the truth of $maxand_{i+2} \geq maxnor_{i+2}$ given $maxand_i \geq maxnor_i$ can also be summarized below. (Here the impossible combinations have been omitted for brevity.)

$maxand_i \geq maxnor_i \Rightarrow$ $\Downarrow and_{i+1}, nor_{i+1}, and_{i+2}, nor_{i+2} \Downarrow$	true	false
none	true	false
and_{i+1}	true	true
nor_{i+1}	false	false
and_{i+2}	true	true
and_{i+2}, and_{i+1}	true	true
and_{i+2}, nor_{i+1}	true	true
nor_{i+2}	false	false
nor_{i+2}, and_{i+1}	false	false
nor_{i+2}, nor_{i+1}	false	false

We see[2]Use of this form of reasoning is justified by the properties of max, that the value of a max expression depends on a single extreme element that each string of input bit pairs is an operator that can do one of three things: it can act like a single pair of bits both of which are true (called $\langle and \rangle$ below, like a single pair of bits both of which are false (called $\langle nor \rangle$), or it can act like a pair one of which is true (called $\langle other \rangle$). The binary operator $\oplus = \lambda_{x,y}$[if $y = \langle and \rangle$ then $\langle and \rangle$ elseif $y = \langle nor \rangle$ then $\langle nor \rangle$ else x] is associative, and that if the identity of this operator is considered to be $\langle other \rangle$ then $maxand_i > maxnor_i = (\oplus_{0 \leq j \leq i} = \langle and \rangle)$. This is precisely what was needed: an operator, amenable to parallel prefix computation, with finite range and domain.

Use of a specification based on this operator will yield a network similar to Figure 3, except that there will only be a single parallel prefix tree, each bit's carry will be used directly rather than computed from the two parallel prefix trees, and (of course) the widths of the data paths and the sized of the nodes will be smaller.

§3.3 Ripple-carry and Bit Serial Circuits

Consider our "standard specification" of Figure 1. If we app the quantifier levelling of Subsection 4.1, we get:

$$\forall\, 0 \leq i \leq n-1$$
$$c_i = a_i \otimes b_i \otimes \max_{j<i}[a_j \wedge b_j] \geq \max_{k<i}[\sim(a_k \vee b_k)]$$

We repeat the reasoning for representing $max_{0 \leq j \leq i+1}[P(x)]$ in terms of $max_{0 \leq j \leq i}[P(x)]$ and $P(i+1)$ (see Subsection 4.2). We also apply the next argument of that Subsection , giving a recurrence for the max... $\geq$ max... expression. That expression has a single free variable, i, and we will call its value $carry_i$.

2.

Consider the recurrence $carry_0$=**false** and $carry_{i+1}=(carry_i \wedge (a_i \vee b_i)) \vee (a_i \wedge b_i)$[3]More precisely, $carry_{i+1}$=**if** $a_i \wedge b_i$ **then true elseif** $\sim (a_i \vee b_i)$ **then false else** $carry_i$. . This leads immediately to the grade school specification of Figure 2.

Using the techniques of TRANSCONS (assigning a processor to each value, developing an interconnection graph, and specifying the appropriate work for each processor), we immediately get the ripple-carry unit shown below.

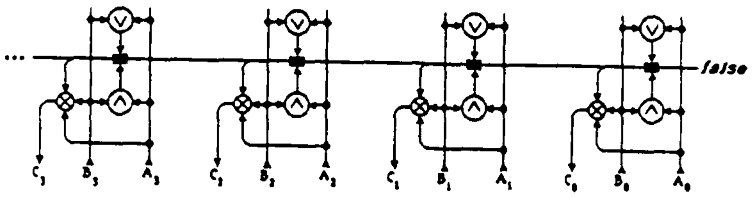

...
false
C_3 B_3 A_3 C_2 B_2 A_2 C_1 B_1 A_1 C_0 B_0 A_0

Figure 5. Ripple Carry Parallel Structure

A technique called *aggregation* [King-83] is applicable. This technique replaces a related series of processing elements by a single element that receives a series of related data. The circuit of Figure 4 is an indexed series of identical modules, and identifying corresponding nodes of the series of modules gives the bit serial addition circuit shown below.

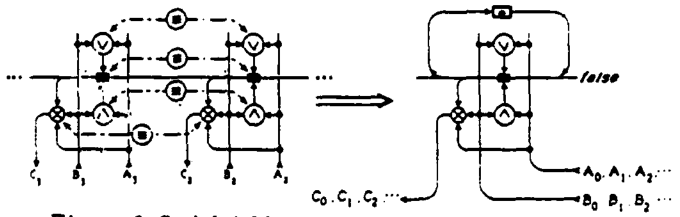


Figure 6. Serial Adder

3†

References

AKS-83]	M. Ajtai, J. Komlós and E. Szemerédi "An $O(n \log n)$ Sorting Network" *Proceedings of the 15th ACM Symposium on Theory of Computing, pp. 1-9,* 1983
'ArmGec-78]	W. Armstrong and J. Gecsei "Adaptation Algorithms for Binary Tree Networks" *University of Montreal Publication 289*, 1978
ArmGec-79]	W. Armstrong J. Gecsei "Architecture of a Tree-Based Image Processor" *Tech Report, University of Montreal, Publication 291*, 1979
AtkHew-77]	R. Atkinson and C. Hewitt "Specification and Proof Techniques for Serializers" *MIT AI Lab Tech Memo 438*, August 1977
Barter-82]	C. Bartet "Policy-Protocol Interaction in Composite Processes" *MIT AI Lab Memo 692*, September 1982
Batcher-68]	K. Batcher "Sorting Networks and their Applications" *AFIPS Spring Joint Computer Conference, pp. 307-314*, 1968
BenKung-79]	J. Bentley and H. Kung "Two Papers on a Tree-Structured Parallel Computer" *Carnage-Mellun University Tech Report CMU-CS-79-142*, September 1979
BSdJ-82]	R. Byrd, S. Smith and S. de Jong "An Actor-Based Programming System" *IBM Research Report #RC 9204 (#40424)*, January 1982
'ChanMis-78]	K. Chandy and J. Misra "Specification, Synthesis, Verification and Performance Analysis of Distributed Programs; a Case Study; Distributed Simulation" *University of Texas, Austin Tech Report TR-86*, November 1978
[ChenMead-82]	M. Chen and C. Mead "Formal Specifications of Concurrent Systems" *Technical report 5042:TR:82, California Institute of Technology*, 1982
Choo-82]	Y. Choo "Hierarchial Nets – A Structured Petri Network Approach to Concurrency" *Cal Tech Report TR:5044:82*, November 1982
Church-51]	A. Church "The Calculi of Lambda-Conversion" *Annals of Mathematical Studies # 6*, Princeton University Press
Clarke-78]	E. Clarke "Concurrent Programs are Easier to Verify than Sequential Programs" *Duke University Tech Report CS-1978-6*, July 1978

[Clinger-81]	W. Clinger "Foundations of Actor Semantics", PhD Thesis , *MIT AI Lab Tech Report AI-TR-633*, May 1981
[CLW-79]	K. Chung, F. Luccio and C. Wong "A Tree Storage Scheme for Magnetic Bubble Memories" *IBM Research Report #RC 8116 (#34797)*, December 1979
[CuiPachl-82]	K. CilJ. Pachl "Folding and Unrolling Systolic Arrays" *University of Waterloo Research Report CS-82-11*, April 1982
[Dennis-75]	J. Dennis "First Version of a Data Flow Procedure Language" *Project MAC, MIT*, May 1975
[Edwards-78]	N. Edwards "Configurable Pipelined Application Logic Systems" *IBM Research Report #RC 7313 (#31451)*, September 1978
[Fich-83]	Faith E. Fich "New Bounds for Parallel Prefix Circuits" *Proceedings of the 15th ACM Symposium on Theory of Computing, pp. 100-109*, 1983
[FishPat-80]	M. Fischer and M. Paterson "Optimal Tree Layout" *University of Washington Tech Report 80-03-02*, February 1980
[GalPaul-83]	Z. Galil and W. Paul "An Efficient General-Purpose Parallel Computer" *Journal of the ACM, vol. 30 #2, pp. 360-387*, April 1983
[Galtieri-80]	C. Galtieri "Architecture for a Consistent Decentralized System" *IBM Research Report #RJ2846(36132)*, June 1980
[Griffiths-75]	P. Griffiths "SYNVER: An Automatic System for the Synthesis and Verification of Synchronous Processes" *Harvard PhD Thesis and Tech Report TR-20-75*, June 1975
[Hack-75]	M. Hack "Decidability Questions for Petri Nets" *PhD Thesis, MIT MAC Tech Report MAC-TR-161*, December 1975
[Hailpern-81]	B. Hailpern "Modular Verification of Concurrent Programs"IBM Research Report #RC 9130 (#39971) , *November 1981*
[Halstead-78]	R. Halstead Jr. "Multiple-Processor Implementations of Message-Passing Systems", Masters Thesis , *MIT Tech Report MIT-LCS-TR-198*, January 1978
[Harbison-80]	S. Harbison "A Computer Architecture for the Dynamic Optimization of High-Level Language Programs" *PhD Thesis, CMU, Tech Report CMU-CS-80-143*, September 1980
[HMS-83]	P. Hochschild, E. W. Mayr, and A. Siegel "Techniques for Solving Graph Problems in Parallel Environments" *Proceedings of the 24th Symposium on Foundations of Computer Science* to appear November 1983
[Kant-79]	Elaine Kant "Efficiency Considerations in Program Synthesis: A Knowledge-Based Approach" , *Ph. D. Thesis, Department of Computer Science, Stanford University*, 1979
[King-83]	R. King "Research on Synthesis of Concurrent Computing Systems" *Proceedings of the 10th Symposium on Computer Architecture, pp. 39-46*, 1983
[KingBrown-83]	R. King and T. Brown "Proposal for Research On Automatic Synthesis

of Tree-Structured Concurrent Computing Systems" , *Kestrel Tech Report #KES.L.83.1*, 1983

[KungLeh-79] H. Kung and P. Lehman "Systolic (VLSI) Arrays for Relational Database Operations" *Carnegie Mellon University Tech Report CMU-CS-80-114*, October 1979

[KungLei-78] H. T. Kung and Charles E. Leiserson "Systolic Arrays for VLSI" *Sparse Matrix Proceedings, 1978*

[LadFish-80] R. Ladner and M. Fischer "Parallel Prefix Computation" *Journal of the ACM, vol. 27 #4, pp. 831-838*, 1980

[Leighton-81] F. T. Leighton "A Layout Strategy for VLSI Which is Provably Good" *Proceedings of the 14th ACM Symposium on Theory of Computing, pp. 85-97*, 1982

[LeiSaxe-81] C. Leiserson and J. Saxe "Optimizing Synchronous Systems" *Proceedings of the 22nd Annual Symposium on the Foundations of Computer Science, Pp. 23-36*, 1981

[LeiSaxe-82] C. Leiserson and J. Saxe "Optimizing Synchronous Systems" *MIT Tech Report MIT/LCS/TM-215*, March 1982

[Lengauer-82] C. Lengauer "A Methodology for Programming with Concurrency" *U. of Toronto Tech Report CSRG-142*, April 1982

[LipVal-81] R. J Lipton and J. Valdes "Census Functions: an Approach to VLSI Upper Bounds" , *Proceedings of the 21st IEEE Symposium on the Foundations of Computer Science, pp. 13-22*, 1981

[Milner-78] R. Milner "Algebras for Communicating Systems" *Tech Report, University of Edinburgh #CSR-25-78*, April 1978

[MirWin-84] W. Miranker and A. Winkler "Spacetime Representation of Computational Structures" *Computing 32*, 1984 Pp. 93-114

[Paige-79] R. Paige "Expression Continuity and the Formal Differentiation of Algorithms" *Technical Report #15, Courant Institute, New York, pp. 269-658*, 1979

[Rambaugh-75] J. Rambaugh "A Parallel Asynchronous Computer Architecture for Data Flow Programs" PhD Thesis, MIT MAC Tech Report MAC-TR-150 , *May 1975*

[Ramchandani-73] C. Ramchandani "Analysis of Asynchronous Concurrent Systems by timed Petri Nets" *PhD Thesis, MIT MAC Tech Report MAC-TR-120*, July 1973

[ReifVal-82] J. Reif and L. Valiant "A Logarithmic Time Sort for Linear Size Networks" , *Harvard Tech Report #TR-13-82*, 1982

[Schwartz-80] J. Schwartz "Ultracomputers" *ACM TOPLAS, vol. 2 #4 pp. 484-521*, October 1980

[Smith-83] D. Smith "Top-Down Synthesis of Simple Divide & Conquer Algorithms" *Tech Report, Naval Postgraduate School, Montery, CA 93940*, November 1983

[Smith-83] D. Smith "Derived Preconditions and Their Use in Program Synthesis" *Tech*

Report, Naval Postgraduate School, Montery, CA 93940, November 1983

[Theriault-82] D. Theriault "A Primer for the Act-1 Language" *MIT AI Lab Memo 672*, April 1982

[Weng-79] K. Weng "An Abstract Implementation for a Generalized Data Flow Language"PhD Thesis, MIT Tech Report MIT-LCS-TR-228 , *May 1979*

[Wolper-82] P. Wolper "Synthesis of Communicating Processes from Temporal Logic Specifications" *Stanford Tech Report STAN-CS-82-925*, August 1982